

第19章 shell 函数

本书目前为止所有脚本都是从头到尾执行。这样做很好，但你也许已经注意到有些脚本段间互相重复。

shell允许将一组命令集或语句形成一个可用块，这些块称为 shell函数。

本章内容有：

- 定义函数。
- 在脚本中使用函数。
- 在函数文件中使用函数。
- 函数举例。

函数由两部分组成：

函数标题。

函数体。

标题是函数名。函数体是函数内的命令集合。标题名应该唯一；如果不是，将会混淆结果，因为脚本在查看调用脚本前将首先搜索函数调用相应的 shell。

定义函数的格式为：

函数名 ()

```
{  
命令1
```

```
...
```

```
}
```

或者

函数名 () {

```
命令1
```

```
...
```

```
}
```

两者方式都可行。如果愿意，可在函数名前加上关键字 `function`，这取决于使用者。

`function` 函数名 ()

```
{ ...  
}
```

可以将函数看作是脚本中的一段代码，但是有一个主要区别。执行函数时，它保留当前 shell和内存信息。此外如果执行或调用一个脚本文件中的另一段代码，将创建一个单独的 shell，因而去除所有原脚本中定义的存在变量。

函数可以放在同一个文件中作为一段代码，也可以放在只包含函数的单独文件中。函数不必包含很多语句或命令，甚至可以只包含一个 `echo`语句，这取决于使用者。

19.1 在脚本中定义函数

以下是一个简单函数

```
hello ()
{
echo "Hello there today's date is `date`"
}
```

所有函数在使用前必须定义。这意味着必须将函数放在脚本开始部分，直至 shell 解释器首次发现它时，才可以使用。调用函数仅使用其函数名即可。上面的例子中，函数名为 `hello`，函数体包含一个 `echo` 语句，反馈当天日期。

19.2 在脚本中使用函数

现在创建函数，观察其在脚本中的用法。

```
$ pg func1
#!/bin/sh
# func1
hello ()
{
echo "Hello there today's date is `date`"
}
```

```
echo "now going to the function hello"
hello
```

```
echo "back from the function"
```

运行脚本，结果为：

```
$ func1
now going to the function hello
Hello there today's date is Sun Jun  6 10:46:59 GMT 1999
back from the function
```

上面例子中，函数定义于脚本顶部。可以在脚本中使用函数名 `hello` 调用它。函数执行后，控制返回函数调用的下一条语句，即反馈语句 `back from the function`。

19.3 向函数传递参数

向函数传递参数就像在一般脚本中使用特殊变量 `$1, $2...$9` 一样，函数取得所传参数后，将原始参数传回 shell 脚本，因此最好先在函数内重新设置变量保存所传的参数。这样如果函数有一点错误，就可以通过已经本地化的变量名迅速加以跟踪。函数里调用参数（变量）的转换以下划线开始，后加变量名，如：`_FILENAME` 或 `_filename`。

19.4 从调用函数中返回

当函数完成处理或希望函数基于某一测试语句返回时，可做两种处理：

1) 让函数正常执行到函数末尾，然后返回脚本中调用函数的控制部分。

2) 使用 `return` 返回脚本中函数调用的下一条语句，可以带返回值。0 为无错误，1 为有错误。

这是可选的，与最后状态命令报表例子极其类似。其格式为：

```
return    从函数中返回， 用最后状态命令决定返回值。  
Return 0  无错误返回。  
Return 1  有错误返回
```

19.5 函数返回值测试

可以直接在脚本调用函数语句的后面使用最后状态命令来测试函数调用的返回值。例如：

```
check_it_is_a_directory $FILENAME    # this is the function call and check  
if [ $? = 0 ]    # use the last status command now to test  
then  
    echo "All is OK"  
else  
    echo " Something went wrong!"  
fi
```

更好的办法是使用 if 语句测试返回 0 或者返回 1。最好在 if 语句里用括号将函数调用括起来以增加可读性。例如：

```
if check_it_is_a_directory $FILENAME; then  
    echo "All is OK"  
    # do something ??  
else  
    echo "Something went wrong !"  
    # do something ??  
fi
```

如果函数将从测试结果中反馈输出，那么使用替换命令可保存结果。函数调用的替换格式为：

```
variable_name=function_name
```

函数 function_name 输出被设置到变量 variable_name 中。

不久我们会接触到许多不同的函数及使用函数的返回值和输出的不同方法。

19.6 在shell中使用函数

当你收集一些经常使用的函数时，可以将之放入函数文件中并将文件载入 shell。

文件头应包含语句 `#!/bin/sh`，文件名可任意选取，但最好与相关任务有某种实际联系。例如，`functions.main`。

一旦文件载入 shell，就可以在命令行或脚本中调用函数。可以使用 `set` 命令查看所有定义的函数。输出列表包括已经载入 shell 的所有函数。

如果要改动函数，首先用 `unset` 命令从 shell 中删除函数，尽管 `unset` 删除了函数以便于此函数对于 shell 或脚本不可利用，但并不是真正的删除。改动完毕后，再重新载入此文件。有些 shell 会识别改动，不必使用 `unset` 命令，但为了安全起见，改动函数时最好使用 `unset` 命令。

19.7 创建函数文件

下面创建包容函数的函数文件并将之载入 shell，进行测试，再做改动，之后再重新载入。

函数文件名为 `functions.main`，内容如下：

```
$ pg functions.main
#!/bin/sh
# functions.main
#
# findit: this is front end for the basic find command
findit() {
# findit
if [ $# -lt 1 ]; then
    echo "usage :findit file"
    return 1
fi
find / -name $1 -print
```

上述脚本本书前面用过，现在将之转化为一个函数。这是一个基本 `find` 命令的前端。如果不加参数，函数将返回 1，即发生错误。注意错误语句中用到了实际函数名，因为这里用 `$0`，shell 将只返回 sh-信息，原因是文件并不是一个脚本文件。这类信息对用户帮助不大。

19.8 定位文件

定位文件格式为：

```
./pathname/filename
```

现在文件已经创建好了，要将之载入 shell，试键入：

```
$ . functions.main
```

如果返回信息 `file not found`，再试：

```
$ . /functions.main
```

此即<点><空格><斜线><文件名>，现在文件应该已载入 shell。如果仍有错误，则应该仔细检查是否键入了完整路径名。

19.9 检查载入函数

使用 `set` 命令确保函数已载入。 `set` 命令将在 shell 中显示所有的载入函数。

```
$ set
USER=dave
findit=()
{
if [ $# -lt 1 ]; then
    echo "usage :findit file";
    return 1;
fi;
find / -name $1 -print
}
...
```

19.10 执行shell函数

要执行函数，简单地键入函数名即可。这里是带有一个参数的 `findit` 函数，参数是某个系统文件。

```
$ findit groups
```

```
/usr/bin/groups
/usr/local/backups/groups.bak
```

19.10.1 删除shell函数

现在对函数做一些改动。首先删除函数，使其对 shell不可利用。使用 unset命令完成此功能。删除函数时unset命令格式为：

```
unset function_name
$ unset findit
```

如果现在键入set命令，函数将不再显示。

19.10.2 编辑shell函数

编辑函数functions.main，加入for循环以便脚本可以从命令行中读取多个参数。改动后函数脚本如下：

```
$ pg functions.main
#!/bin/sh
findit()
{
# findit
if [ $# -lt 1 ]; then
    echo "usage :findit file"
    return 1
fi
for loop
do
    find / -name $loop -print
done
}
```

再次定位函数

```
$ . /functions.main
```

使用set命令查看其是否被载入，可以发现 shell正确解释for循环以接受所有输入参数。

```
$ set
findit=()
{
if [ $# -lt 1 ]; then
    echo "usage :`basename $0` file";
    return 1;
fi;
for loop in "$@";
do
    find / -name $loop -print;
done
}
...
```

现在执行改动过的findit函数，输入两个参数：

```
$ findit LPS0.doc passwd
/usr/local/accounts/LPS0.doc
/etc/passwd
...
```

19.10.3 函数举例

既然已经学习了函数的基本用法，现在就用它来做一些工作。函数可以节省大量的编程时间，因为它是可重用的。

1. 变量输入

以下脚本询问名，然后是姓。

```
$ pg func2
#!/bin/sh
# func2
echo -n "What is your first name : "
read F_NAME
echo -n "What is your surname : "
read S_NAME
```

要求输入字符必须只包含字母。如果不用函数实现这一点，要写大量脚本。使用函数可以将重复脚本删去。这里用 awk 语言测试字符。以下是取得只有小写或大写字符的测试函数。

```
char_name()
{
# char_name
# to call: char_name string
# assign the argument across to new variable
_LETTERS_ONLY=$1
# use awk to test for characters only !
_LETTERS_ONLY=`echo $1|awk '{if($0~/[Aa-z A-Z]/) print "1"}'`
if [ "$_LETTERS_ONLY" != "" ]
then
# oops errors
return 1
else
# contains only chars
return 0
fi
}
```

首先设置变量 \$1 为一有意义的名字，然后用 awk 测试整个传送记录只包含字母，此命令输出（1 为非字母，空为成功）保存在变量 _LETTERS_ONLY 中。

然后执行变量测试，如果为空，则为成功，如果有值，则为错误。基于此项测试，返回码然后被执行。在对脚本的函数调用部分进行测试时，使用返回值会使脚本清晰易懂。

使用 if 语句格式测试函数功能：

```
if char_name $F_NAME; then
echo "OK"
else
echo "ERRORS"
fi
```

如果有错误，可编写一个函数将错误反馈到屏幕上。

```
name_error()
# name_error
# display an error message
{
echo " $@ contains errors, it must contain only letters"
}
```

函数name_error用于显示所有无效输入错误。使用特殊变量 \$@ 显示所有参数，这里为变量F_NAME和S_NAME值。完成脚本如下：

```
$ pg func2
!/bin/sh
char_name()
# char_name
# to call: char_name string
# check if $1 does indeed contain only characters a-z,A-Z
{
# assign the argument across to new variable
_LETTERS_ONLY=$1
_LETTERS_ONLY=`echo $1|awk '{if($0~/^[a-zA-Z]/) print "1"}'`
if [ "$_LETTERS_ONLY" != "" ]
then
# oops errors
return 1
else
# contains only chars
return 0
fi
}

name_error()
# display an error message
{
echo " $@ contains errors, it must contain only letters"
}

while :
do
echo -n "What is your first name : "
read F_NAME
if char_name $F_NAME
then
# all ok breakout
break
else
name_error $F_NAME
fi
done

while :
do
echo -n "What is your surname : "
read S_NAME
if char_name $S_NAME
then
# all ok breakout
break
else
name_error $S_NAME
fi
done
```

注意每个输入的while循环，这将确保不断提示输入直至为正确值，然后跳出循环。当然，

实际脚本拥有允许用户退出循环的选项，可使用适当的游标，正像控制 0 长度域一样。

```
$ func2
What is your first name :Davi2d
Davi2d contains errors, it must contain only letters
What is your first name :David
What is your surname :Tansley1
Tansley1 contains errors, it must contain only letters
What is your surname :Tansley
```

2. echo问题

echo语句的使用类型依赖于使用的系统是 LINUX、BSD还是系统 V，本书对此进行了讲解。

下面创建一个函数决定使用哪种 echo语句。

使用echo时，提示应放在语句末尾，以等待从 read命令中接受进一步输入。

LINUX和BSD为此使用echo命令-n选项。

以下是LINUX (BSD) echo语句实例，这里提示放于echo后面：

```
$ echo -n "Your name : "
Your name : {{?}}
```

系统V使用\C保证在末尾提示：

```
$ echo "Your name : \c"
Your name : □
```

在echo语句开头LINUX使用-e选项反馈控制字符。其他系统使用反斜线保证 shell获知控制字符的存在。

有两种方法测试echo语句类型，下面讲述这两种方法，这样，就可以选择使用其中一个。

第一种方法是在echo语句里包含测试控制字符。如果键入\007和一个警铃，表明为系统V，如果只键入\007，显示为LINUX。

以下为第一个控制字符测试函数。

```
uni_prompt ()
# uni_prompt
# universal echo
{
if [ `echo "\007"` = "\007" ] >/dev/null 2>&1
# does a bell sound or are the characters just echoed??
then
# characters echoed, it's LINUX /BSD
echo -e -n "$@"
else
# it's System V
echo "$@\c"
fi
}
```

注意这里又用到了特殊变量\$@以反馈字符串，要在脚本中调用上述函数，可以使用：

```
uni_prompt "\007 there goes the bell ,What is your name:"
```

这将发出警报并反馈 ' What is your name: '，并在行尾显示字符串。如果在末尾出现字符，则为系统V版本，否则为LINUX/BSD版本。

第二种方法使用系统V \c测试字母z是否悬于行尾。

```
uni_prompt ()
```



```
# uni_prompt
# universal prompt
{
if [ `echo "Z\c"` = "Z" ] >/dev/null 2>&1
# echo any chracter out, does it hang on to the end of line ???
then
# yes, it's System V
echo "$@\c"
else
# No, it's LINUX, BSD
echo -e -n "$@"
fi
}
```

要在脚本中调用上述函数，可以使用：

```
uni_prompts "\007 there goes the bell, What is your name:"
```

使用两个函数中任意一个，并加入一小段脚本：

```
uni_prompt "\007 There goes the bell, What is your name :"  
read NAME
```

将产生下列输出：

```
There goes the bell, What is your name:
```

3. 读单个字符

在菜单中进行选择时，最麻烦的工作是必须在选择后键入回车键，或显示“press any key to continue”。可以使用dd命令解决不键入回车符以发送击键序列的问题。

dd命令常用于对磁带或一般的磁带解压任务中出现的数问题提出质疑或转换，但也可用于创建定长文件。下面创建长度为1兆的文件myfile。

```
dd if=/dev/zero of=myfile count=512 bs=2048
```

dd命令可以翻译键盘输入，可被用来接受多个字符。这里如果只要一个字符，dd命令需要删除换行字符，这与用户点击回车键相对应。dd只送回车前一个字符。在输入前必须使用stty命令将终端设置成未加工模式，并在dd执行前保存设置，在dd完成后恢复终端设置。

函数如下：

```
read_a_char()
# read_a_char
{
# save the settings
SAVEDSTTY=`stty -g`
# set terminal raw please
stty cbreak
# read and output only one character
dd if=/dev/tty bs=1 count=1 2> /dev/null
# restore terminal and restore stty
stty -cbreak
stty $SAVEDSTTY
}
```

要调用函数，返回键入字符，可以使用命令替换操作，例子如下：

```
echo -n "Hit Any Key To Continue"  
character=`read_a_char`  
echo " In case you are wondering you pressed $character"
```

4. 测试目录存在

拷贝文件时，测试目录是否存在是常见的工作之一。以下函数测试传递给函数的文件名是否是一个目录。因为此函数返回时带有成功或失败取值，可用 if 语句测试结果。

函数如下：

```
is_it_a_directory()
{
# is_it_a_directory
# to call: is_it_a_directory directory_name
if [ $# -lt 1 ]; then
    echo "is_it_a_directory: I need an argument"
    return 1
fi
# is it a directory ?
_DIRECTORY_NAME=$1
if [ ! -d $_DIRECTORY_NAME ]; then
    # no it is not
    return 1
else
    # yes it is
    return 0
fi
}
```

要调用函数并测试结果，可以使用：

```
echo -n "enter destination directory : "
read DIREC
if is_it_a_directory $direc;
then :
else
    echo "$DIREC does not exist, create it now ? [y..n]"
    # commands go here to either create the directory or exit
    ...
    ...
fi
```

5. 提示Y或N

许多脚本在继续处理前会发出提示。大约可以提示以下动作：

- 创建一个目录。
- 是否删除文件。
- 是否后台运行。
- 确认保存记录。

等等

以下函数是一个真正的提示函数，提供了显示信息及缺省回答方式。缺省回答即用户按下回车键时采取的动作。case语句用于捕获回答。

```
continue_prompt()
# continue_prompt
to call: continue_prompt "string to display" default_answer
{
_STR=$1
_DEFAULT=$2
```

```
# check we have the right params
if [ $# -lt 1 ]; then
    echo "continue_prompt: I need a string to display"
    return 1
fi
# loop forever
while :
do
    echo -n "$_STR [Y..N] [$DEFAULT]:"
    read _ANS
    # if user hits return set the default and determine the return value,
    # that's a : then a <space> then $
    : ${_ANS:= $DEFAULT}
    if [ "$_ANS" = "" ]; then
        case $_ANS in
            Y) return 0 ;;
            N) return 1 ;;
        esac
    fi
    # user has selected something
    case $_ANS in
        y|Y|Yes|YES)
            return 0
            ;;
        n|N|No|NO)
            return 1
            ;;
        *) echo "Answer either Y or N, default is $DEFAULT"
            ;;
    esac
    echo $_ANS
done
}
```

要调用上述函数，须给出显示信息或参数 \$1，或字符串变量。缺省回答 Y或N方式也必须指定。

以下是几种函数 continue_prompt 的调用格式。

```
if continue_prompt "Do you want to delete the var filesystem" "N"; then
    echo "Are you nuts!!"
else
    echo "Phew !, what a good answer"
fi
```

在脚本中加入上述语句，给出下列输入：

```
Do you really want to delete the var filesystem [Y..N] [N]:
phew !!
```

```
Do you really want to delete the var filesystem [Y..N] [N]:y
are you nuts..
```

现在可以看出为什么函数要有指定的缺省回答。

以下是函数调用的另一种方式：

```
if continue_prompt "Do you really want to print this report" "Y"; then
    lpr report
else:
    ..
```

```
fi
```

也可以使用字符串变量\$1调用此函数：

```
if continue_prompt $1 "Y"; then
    \pr report
else:
fi
```

6. 从登录ID号中抽取信息

当所在系统很庞大，要和一登录用户通信时，如果忘了用户的全名，这是很讨厌的事。比如有时你看到用户锁住了一个进程，但是它们的用户 ID号对你来说没有意义，因此必须要用grep passwd文件以取得用户全名，然后从中抽取可用信息，向其发信号，让其他用户开锁。以下函数用于从grep /etc/passwd命令抽取用户全名。

本系统用户全名位于passwd文件域5中，用户的系统可能不是这样，这时必须改变其域号以匹配passwd文件。

这个函数需要一个或多个用户ID号作为参数。它对密码文件进行grep操作。

函数脚本如下：

```
whois()
# whois
# to call: whois userid
{
# check we have the right params
if [ $# -lt 1 ]; then
    echo "whois : need user id's please"
    return 1
fi

for loop
do
    _USER_NAME=`grep $loop /etc/passwd | awk -F: '{print $4}'`
    if [ "$_USER_NAME" = "" ]; then
        echo "whois: Sorry cannot find $loop"
    else
        echo "$loop is $_USER_NAME"
    fi
done
}
```

以下为whois函数调用方式：

```
$ whois dave peters superman
dave is David Tansley - admin accts
peter is Peter Stromer - customer services
whois: Sorry cannot find superman
```

7. 列出文本文件行号

在vi编辑器中，可以列出行号来进行调试，但是如果打印几个带有行号的文件，必须使用nl命令。以下函数用nl命令列出文件行号。原始文件中并不带有行号。

```
number_file()
# number_file
# to call: number_file filename
{
    _FILENAME=$1
```

```
# check we have the right params
if [ $# -ne 1 ]; then
    echo "number_file: I need a filename to number"
    return 1
fi

loop=1
while read LINE
do
    echo "$loop: $LINE"
    loop=`expr $loop + 1`
done < $_FILENAME
}
```

要调用number_file函数，可用一个文件名做参数，或在 shell中提供一文件名，例如：

```
$ number_file myfile
```

也可以在脚本中这样写或用：

```
number_file $1
```

输出如下：

```
$ number_file /home/dave/file_listing
1: total 105
2: -rw-r--r--  1 dave      admin      0 Jun  6 20:03 DT
3: -rw-r--r--  1 dave      admin    306 May 23 16:00 LPSO.AKS
4: -rw-r--r--  1 dave      admin    306 May 23 16:00 LPSO.AKS.UC
5: -rw-r--r--  1 dave      admin    324 May 23 16:00 LPSO.MBB
6: -rw-r--r--  1 dave      admin    324 May 23 16:00 LPSO.MBB.UC
7: -rw-r--r--  1 dave      admin    315 May 23 16:00 LPSO.MKQ
...
...
```

8. 字符串大写

有时需要在文件中将字符串转为大写，例如在文件系统中只用大写字符创建目录或在有效的文本域中将输入转换为大写数据。

以下是相应功能函数，可以想像要用到 tr命令：

```
str_to_upper ()
# str_to_upper
# to call: str_to_upper $1
{
    _STR=$1
    # check we have the right params
    if [ $# -ne 1 ]; then
        echo "number_file: I need a string to convert please"
        return 1
    fi
    echo "$@" |tr '[a-z]' '[A-Z]'
}
```

变量upper保存返回的大写字符串，注意这里用到特定参数 \$@来传递所有参数。

str_to_upper可以以两种方式调用。在脚本中可以这样指定字符串。

```
UPPER=`str_to_upper "documents.live"`
echo $upper
```

或者以函数输入参数\$1的形式调用它。

```
UPPER=`str_to_upper $1`
echo $UPPER
```

两种方法均可用替换操作以取得函数返回值。

9. is_upper

虽然函数str_to_upper做字符串转换，但有时在进一步处理前只需知道字符串是否为大写。

is_upper实现此功能。在脚本中使用if语句决定传递的字符串是否为大写。

函数如下：

```
is_upper()
# is_upper
# to call: is_upper $1
{
# check we have the right params
if [ $# -ne 1 ]; then
    echo "is_upper: I need a string to test OK"
    return 1
fi
# use awk to check we have only upper case
_IS_UPPER=`echo $1|awk '{if($0~/[A-Z]/) print "1"}'`
if [ "$_IS_UPPER" != "" ]
then
    # no, they are not all upper case
    return 1
else
    # yes all upper case
    return 0
fi
}
```

要调用is_upper，只需给出字符串参数。以下为其调用方式：

```
echo -n "Enter the filename : "
read FILENAME
if is_upper $FILENAME; then
    echo "Great it's upper case"
    # let's create a file maybe ??
else
    echo "Sorry it's not upper case"
    # shall we convert it anyway using str_to_upper ???
fi
```

要测试字符串是否为小写，只需在函数 is_upper中替换相应的 awk语句即可。此为is_lower。

```
_IS_LOWER=`echo $1|awk '{if($0~/[a-z]/) print "1"}'`
```

10. 字符串小写

现在实现此功能，因为已经给出了 str_to_upper，最好相应给出str_to_lower。函数工作方式与前面一样。

函数如下：

```
str_to_lower ()
# str_to_lower
# to call: str_to_lower $1
{
```

```
# check we have the right params
if [ $# -ne 1 ]; then
    echo "str_to_lower: I need a string to convert please"
    return 1
fi
echo "$@" | tr '[:upper:]' '[:lower:]'
}
```

变量 LOWER 保存最近返回的小写字符串。注意用到特定参数 `$@` 传递所有参数。

str_to_lower调用方式也分为两种。可以在脚本中给出字符串：

```
LOWER=`str_to_lower "documents.live"`
echo $LOWER
```

或在函数中用参数代替字符串：

```
LOWER=`str_to_upper $1`
echo $LOWER
```

11. 字符串长度

在脚本中确认域输入有效是常见的任务之一。确认有效包括许多方式，如输入是否为数字或字符；域的格式与长度是否为确定形式或值。

假定脚本要求用户交互输入数据到名称域，你会想控制此域包含字符数目，比如人名最多为20个字符。有可能用户输入超过 50个字符。以下函数实施控制功能。需要向函数传递两个参数，实际字符串和字符串最大长度。

函数如下：

```
check_length()
# check_length
# to call: check_length string max_length_of_string
{
    _STR=$1
    _MAX=$2
    # check we have the right params
    if [ $# -ne 2 ]; then
        echo "check_length: I need a string and max length the string should be"
        return 1
    fi
    # check the length of the string
    _LENGTH=`echo $_STR | awk '{print length($0)}'`
    if [ "$_LENGTH" -gt "$_MAX" ]; then
        # length of string is too big
        return 1
    else
        # string is ok in length
        return 0
    fi
}
```

调用函数 check_length：

```
$ pg test_name
# !/bin/sh
# test_name
while :
do
```

循环持续直到输入到变量 NAME 的数据小于最大字符长度，这里指定为 10，break 命令然跳出循环。

```
$ val_max
Enter your FIRST name :Pertererxxxxxxxxxxxxx
The name field is too long 10 characters max
Enter your FIRST name :Peter
```

以下是wc命令的缺点举例（也可以称为特征之一）

运行上述脚本（其中 为空格）

12. chop

MYDOCUMENT.DOC 10

Chop 函数如下：

```
chop()
# chop
# to call: chop string how_many_chars_to_chop
{
  _STR=$1
  _CHOP=$2
  # awk's substr starts at 0, we need to increment by one
  # to reflect when the user says (ie) 2 chars to be chopped it will be 2
  # chars off
  # and not 1
  CHOP=`expr $_CHOP + 1`

  # check we have the right params
  if [ $# -ne 2 ]; then
    echo "check length: I need a string and how many characters to chop"
```



```

    return 1
fi
# check the length of the string first
# we can't chop more than what's in the string !!
_LENGTH=`echo $_STR |awk '{print length($0)}'`
if [ "$_LENGTH" -lt "$_CHOP" ]; then
    echo "Sorry you have asked to chop more characters than there are in
        the string"
    return 1
fi
echo $_STR |awk '{print substr($1,'$_CHOP')}'
}

```

删除后字符串保存于变量 CHOPPED 中，使用下面方法调用 chop 函数：

```

CHOPPED=`chop "Honeysuckle" 5`
echo $CHOPPED
suckle

```

或者：

```

echo -n "Enter the Filename : "
read FILENAME
CHOPPED=`chop $FILENAME 1`
# the first character would be chopped off !

```

13. MONTHS

产生报表或创建屏幕显示时，为方便起见有时要快速显示完整月份。函数 months，接受月份数字或月份缩写作为参数，返回完整月份。

例如，传递3或者03可返回March。函数如下：

```

months()
{
# months
_MONTH=$1
# check we have the right params
if [ $# -ne 1 ]; then
    echo "months: I need a number 1 to 12 "
    return 1
fi

```

```

case $_MONTH in
1|01|Jan)_FULL="January" ;;
2|02|Feb)_FULL="February" ;;
3|03|Mar)_FULL="March";;
4|04|Apr)_FULL="April";;
5|05|May)_FULL="May";;
6|06|Jun)_FULL="June";;
7|07|Jul)_FULL="July";;
8|08|Aug)_FULL="August";;
9|10|Sep|Sept)_FULL="September";;
10|Oct)_FULL="October";;
11|Nov)_FULL="November";;
12|Dec)_FULL="December";;
*) echo "months: Unknown month"
return 1
;;

```

```
esac
echo $_FULL
}
```

用下面方法调用函数 months

```
months 04
```

上面例子显示 April，脚本中使用：

```
MY_MONTH=`months 06`
echo "Generating the Report for Month End $MY_MONTH"
...
```

返回月份 June。

19.10.4 将函数集中在一起

本章目前讲到的函数没有一定的顺序。这些例子只表明函数不一定很长或不一定为一些复杂的脚本。

本书许多函数脚本简单实用，并不需要任何新的后备知识。这些函数只是防止重复输入脚本，实际上这就是函数的基本功能。

本章开始部分，讲到怎样在 shell 中使用函数。第一次使用函数时，也许要花一段时间才能理解其返回值的用法。

本章讲到了几种不同的调用函数及其返回值的方法。如果遇到问题，查看一下实例返回值及其测试方法即可。

以下是一些小技巧。测试函数时，首先将其作为代码测试，当结果满意时，再将其转换为函数，这样做可以节省大量的时间。

19.11 函数调用

本章最后讲述使用函数的两种不同方法：从原文件中调用函数和使用脚本中的函数。

19.11.1 在脚本中调用函数

要在脚本中调用函数，首先创建函数，并确保它位于调用之前。以下脚本使用了两个函数。此脚本前面提到过，它用于测试目录是否存在。

```
$ pg direc_check
#!/bin/sh
# function file
is_it_a_directory()
{
# is_it_a_directory
# to call: is_it_a_directory directory_name
_DIRECTORY_NAME=$1
if [ $# -lt 1 ]; then
echo "is_it_a_directory: I need a directory name to check"
return 1
fi
# is it a directory ?
if [ ! -d $_DIRECTORY_NAME ]; then
return 1
```

```

else
    return 0
fi
}
#-----
error_msg()
{
# error_msg
# beeps; display message; beeps again!
echo -e "\007"
echo $@
echo -e "\007"
    return 0
}
}

### END OF FUNCTIONS

echo -n "enter destination directory : "
read DIREC
if is_it_a_directory $DIREC
then :
else
    error_msg "$DIREC does not exist...creating it now"
    mkdir $DIREC > /dev/null 2>&1
    if [ $? != 0 ]
    then
        error_msg "Could not create directory:: check it out!"
        exit 1
    else :
    fi
fi # not a directory
echo "extracting files..."

```

上述脚本中，两个函数定义于脚本开始部分，并在脚本主体中调用。所有函数都应该在任何脚本主体前定义。注意错误信息语句，这里使用函数 `error_msg` 显示错误，反馈所有传递到该函数的参数，并加两声警报。

19.11.2 从函数文件中调用函数

前面讲述了怎样在命令行中调用函数，这类函数通常用于系统报表功能。

现在再次使用上面的函数，但是这次将之放入函数文件 `functions.sh` 里。sh 意即 shell 脚本。

```

$ pg functions.sh
#!/bin/sh
# functions.sh
# main script functions
is_it_a_directory()
{
# is_it_a_directory
# to call: is_it_a_directory directory_name
#
if [ $# -lt 1 ]; then
    echo "is_it_a_directory: I need a directory name to check"
    return 1

```

```

fi
# is it a directory ?
DIRECTORY_NAME=$1
if [ ! -d $DIRECTORY_NAME ]; then
    return 1
else
    return 0
fi
}

#-----

error_msg()
{
echo -e "\007"
echo $@
echo -e "\007"
return 0
}

```

现在编写脚本就可以调用 functions.sh 中的函数了。注意函数文件在脚本中以下述命令格式定位：

```
.\<path to file>
```

使用这种方法不会创建另一个 shell，所有函数均在当前 shell 下执行。

```

$ pg direc_check
!/bin/sh
# direc_check
# source the function file functions.sh
# that's a <dot><space><forward slash>
. /home/dave/bin/functions.sh

# now we can use the function(s)

echo -n "enter destination directory : "
read DIREC
if is_it_a_directory $DIREC
then :
else
    error_msg "$DIREC does not exist...creating it now"
    mkdir $DIREC > /dev/null 2>&1
    if [ $? != 0 ]
    then
        error_msg "Could not create directory:: check it out!"
        exit 1
    else :
        fi
fi # not a directory
echo "extracting files..."

```

运行上述脚本，可得同样输出结果，好像函数在脚本中一样。

```

$ direc_check
enter destination directory :AUDIT
AUDIT does not exist...creating it now
extracting files...

```

19.12 定位文件不只用于函数

定位文件不只针对于函数，也包含组成配置文件的全局变量。

假定有两个备份文件备份同一系统的不同部分。最好让它们共享一个配置文件。为此需要在一个文件里创建用户变量，然后将一个备份脚本删除后，可以载入这些变量以获知用户在备份开始前是否要改变其缺省值。有时也许要备份到不同的媒体中。

当然这种方法可用于共享配置以执行某一过程的任何脚本。下面的例子中，配置文件backfunc包含一些备份脚本所共享的缺省环境。文件如下：

```
$ pg backfunc
#!/bin/sh
# name: backfunc
# config file that holds the defaults for the archive systems
_CODE="comet"
_FULLBACKUP="yes"
_LOGFILE="/logs/backup/"
_DEVICE="/dev/rmt/0n"
_INFORM="yes"
_PRINT_STATS="yes"
```

缺省文件很清楚，第1域_CODE包含一个脚本关键字。要查看并且改变缺省值，用户必须首先输入匹配_CODE取值的脚本，即“comet”。

以下脚本要求输入密码，成功后显示缺省配置。

```
$ pg readfunc
#!/bin/sh
# readfunc

if [ -r backfunc ]; then
    # source the file
    . /backfunc
else
    echo "`basename $0` cannot locate backfunc file"
fi

echo -n "Enter the code name : "
# does the code entered match the code from backfunc file ???
if [ "${CODE}" != "${_CODE}" ]; then
    echo "Wrong code...exiting..will use defaults"
    exit 1
fi

echo " The environment config file reports"
echo "Full Backup Required      : $_FULLBACKUP"
echo "The Logfile Is              : $_LOGFILE"
echo "The Device To Backup To is   : $_DEVICE"
echo "You Are To Be Informed by Mail : $_INFORM"
echo "A Statistic Report To Be Printed: $_PRINT_STATS"
```

脚本运行时，首先要求输入脚本。脚本匹配后，可以查看缺省值。然后就可以编写脚本让用户改变缺省值。

```
$ readback
Enter the code name :comet
```

```
The environment config file reports
Full Backup Required      : yes
The Logfile Is           : /logs/backup/
The Device To Backup To is : /dev/rmt/0n
You Are To Be Informed by Mail : yes
A Statistic Report To Be Printed: yes
```

19.13 小结

使用函数可以节省大量的脚本编写时间。创建可用和可重用的脚本很有意义，可以使主脚本变短，结构更加清晰。

当创建了许多函数后，将之放入函数文件里，然后其他脚本就可以使用这些函数了。